

# A Hybrid Graph Representation for Exact Graph Algorithms

Faisal N. Abu-Khzam<sup>1</sup>, Karim A. Jahed<sup>1</sup>, and Amer E. Mouawad<sup>2</sup>

<sup>1</sup> Department of Computer Science and Mathematics  
Lebanese American University, Beirut, Lebanon.  
{faisal.abukhzam, karim.jahed}@lau.edu.lb

<sup>2</sup> David R. Cheriton School of Computer Science  
University of Waterloo, Ontario, Canada.  
{aabdomou}@uwaterloo.ca

**Abstract.** Many exact search algorithms for  $\mathcal{NP}$ -hard graph problems adopt the old Davis-Putman branch-and-reduce paradigm. The performance of these algorithms often suffers from the increasing number of graph modifications, such as vertex/edge deletions, that reduce the problem instance and have to be “taken back” frequently during the search process. We investigate practical implementation-based aspects of exact graph algorithms by providing a simple hybrid graph representation that trades space for time to address the said take-back challenge. Experiments on three well studied problems show consistent significant improvements over classical methods.

## 1 Introduction

Despite their super-polynomial asymptotic running times, exact and parameterized algorithms for  $\mathcal{NP}$ -hard graph problems have recently gained great momentum. This could be attributed to a number of facts including the hardness of reasonable polynomial-time approximations as well as the emergence of parameterized complexity theory. A great majority of exact graph algorithms adopt the classical search-tree based recursive backtracking method. The usual goal is to achieve the least-possible asymptotic worst-case running time. While this is of some importance from a theoretical standpoint, the practical significance remains a major challenge.

In general, search-tree based backtracking algorithms for graph problems employ reduction and pruning procedures as well as actions associated with branching decisions that have to be taken, then (frequently) taken back, at every search-tree node. Such algorithms are often described by simple pseudo-code, but their implementation could be more sophisticated, mainly due to the book-keeping needed to be able to undo any graph modifications that result from reduction procedures. A main implementation challenge, therefore, is to reduce the additional cost of *undo* operations.

---

A preliminary version of a portion of this paper was presented at the 4<sup>th</sup> International Frontiers in Algorithmics Workshop, Wuhan, China, 2010.

Normally, graph algorithms are implemented with the adjacency list or the adjacency matrix graph representation. In this paper, we show how a simple combination of the two representations can be used to facilitate the *undo* of many basic graph operations, especially deletions, thereby improving the running time of recursive backtracking (search) algorithms. Generally, every operation that can be taken back is pushed onto a stack and later popped out and performed in reverse. We refer to this action by *explicit-undo*. Our implicit-undo objective is achieved via a highly efficient use of a processor’s control stack only.

In addition to effective backtracking, the hybrid representation combines the advantage of constant time adjacency-queries in adjacency-matrices and the advantage of efficient neighborhood traversal in adjacency-lists. This method works well on simple unweighted graphs and may be extended to weighted graphs. Compared to the use of an adjacency list, the hybrid representation yields improved running times for vertex and edge deletion, permanent edge addition, as well as computing common neighbors.

Despite its simplicity, the introduced method has a surprising great impact on the implementation of any recursive backtracking algorithm. This is shown via experiments conducted on several implementations of known algorithms, using different techniques that were developed and compared for three well-known graph problems: DOMINATING SET, VERTEX COVER, and CLUSTER EDITING. The running times of each algorithm is improved by a factor of at least two on every run, and in some cases the time was reduced from days to minutes.

The paper is organized as follows. First, some background material is provided in Section 2. The hybrid graph representation is presented and discussed in Section 3. In Section 4, we describe the effect of the hybrid method on the running time of common graph modification operations. Section 5 is devoted to our experimental studies and results, and we close with some concluding remarks in Section 6.

## 2 Background

For the sake of completeness, we provide a quick overview of basic graph representation methods while describing some notation and terminology adopted in the paper. An  $n$ -vertex graph  $G = (V, E)$  is usually represented using one of two data structures: adjacency matrices (AM) or adjacency lists (AL). In this paper, we make use of a degrees’ array to keep track of active vertices and the current cardinalities of their neighborhoods. When using AM, neighborhood traversal takes  $\Omega(n)$  time. This is reduced to  $\mathcal{O}(\Delta(G))$  time, where  $\Delta(G)$  is the maximum degree of  $G$ , if we use AL instead. On the other hand, checking if two vertices are adjacent requires  $\mathcal{O}(\Delta(G))$  time in AL and  $\mathcal{O}(1)$  time in AM.

A vertex cover of a graph  $G$  is a set of vertices whose complement induces an edgeless subgraph. In the (parameterized) VERTEX COVER problem, or VC for short, we are given a graph  $G = (V, E)$ , together with a positive integer  $k$ , and we are asked to find a set  $C$  of cardinality  $k$  such that  $C \subseteq V$  and the subgraph induced by  $V \setminus C$  is edgeless. The current fastest worst-case VC algorithm runs

in  $\mathcal{O}(1.2738^k n^{\mathcal{O}(1)})$  time [9]. An optimization algorithm for VC can be obtained by obvious modifications to the parameterized algorithm of [9], or using the MAXIMUM INDEPENDENT SET algorithm from [12].

For comparison purposes, four versions were implemented for VC:

- AL\_VC\_OPT: an optimization version using the adjacency-lists representation, based on simple modifications of the parameterized VC algorithm;
- HYBRID\_VC\_OPT: an optimization version using the hybrid graph representation;
- HYBRID\_VC\_PARM: a parameterized version using the hybrid graph representation but not taking advantage of the *folding* technique [8];
- HYBRID\_VCF\_PARM: a parameterized version using the hybrid graph representation and modified for fast edge-contraction operations.

In the DOMINATING SET problem, henceforth DS, we are given an  $n$ -vertex graph  $G = (V, E)$ , and we are asked to find a set  $D \subset V$  of smallest possible cardinality such that every vertex of  $G$  is either in  $D$  or adjacent to some vertex in  $D$ . DS has received great attention, being a classical  $\mathcal{NP}$ -hard graph optimization problem with many logistical applications.

Until 2004, the best algorithm for DS was still the trivial  $\mathcal{O}^*(2^n)$  enumeration<sup>3</sup>. In that same year, two algorithms were independently published breaking the  $\mathcal{O}^*(2^n)$  barrier [13, 15]. The best worst-case algorithm was presented by Grandoni with a running time in  $\mathcal{O}^*(1.8019^n)$  [15]. Using measure-and-conquer, a bound of  $\mathcal{O}^*(1.5137^n)$  was obtained on the running time of Grandoni’s algorithm [11]. This was later improved to  $\mathcal{O}^*(1.5063)$  in [19] and the current best worst-case algorithm can be found in [18] where a general algorithm for counting minimum dominating sets in  $\mathcal{O}^*(1.5048)$  time is also presented.

For our experimental work, we implemented two versions of the algorithm of [11] where DS is solved by reduction to MINIMUM SET COVER:

- AL\_DS\_OPT: optimization version using the adjacency-lists representation;
- HYBRID\_DS\_OPT: optimization version using the hybrid graph representation.

Our third example is the CLUSTER EDITING problem, which takes a graph  $G$  and a positive integer  $k$  as input and asks whether deleting or adding a total of at most  $k$  edges yields a transitive graph, i.e., a disjoint union of cliques. There is a long sequence of fixed-parameter algorithms for this problem, all are mainly based on dealing with *conflict triples*, which are nothing but induced paths of length two (see [4, 3, 5, 7, 10, 14, 16]). In short, if the graph has an induced path  $x - y - z$ , then either add the edge  $xz$  or delete one of the two other edges. The corresponding branching algorithm runs in  $\mathcal{O}^*(3^k)$  time and can be improved via reduction and pruning methods. Recent versions of the problem were also shown to yield better running times by using multiple parameters [1, 17]. In this paper we use the main (generic) algorithm for CLUSTER EDITING based on

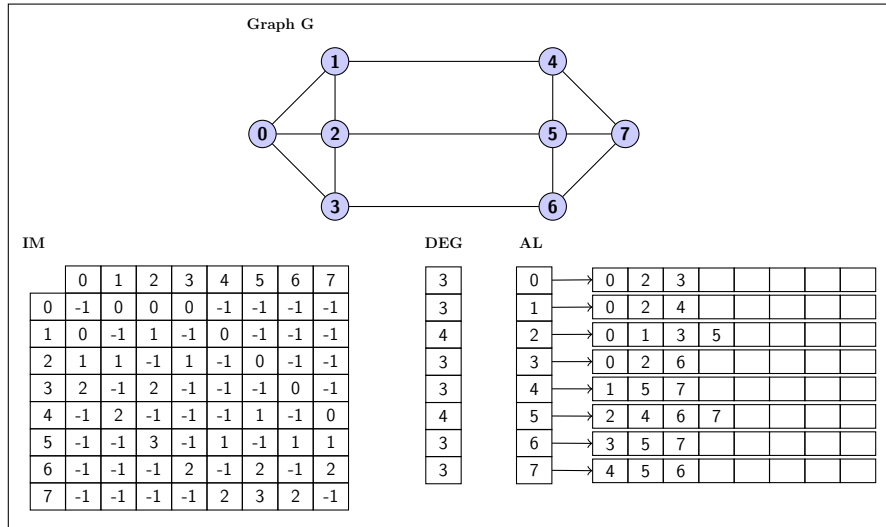
---

<sup>3</sup> Throughout this paper we use the modified big-Oh notation that suppresses all polynomially bounded factors. For functions  $f$  and  $g$  we say  $f(n) \in \mathcal{O}^*(g(n))$  if  $f(n) \in \mathcal{O}(g(n) \text{poly}(n))$ , where  $\text{poly}(n)$  is a polynomial.

branching on conflict triples, with only simple reduction and pruning. The main purpose is not to implement a fastest possible algorithm, but to test the effect of the various operations handled via the hybrid representation, especially edge addition/deletion.

### 3 The Hybrid Graph Representation

The hybrid graph representation is best described using a simple example such as the one given in Figure 1. The adjacency list of a vertex  $v$  is stored in an array denoted by  $AL[v]$ . Accordingly,  $AL[v][i]$  holds the index of the  $i^{th}$  vertex in the list of neighbors of  $v$ . The adjacency matrix, denoted henceforth by  $IM$ , is used as an index table for the adjacency list as follows: the entry  $IM[u][v]$  is equal to the index of  $u$  in  $AL[v]$ .  $IM[u][v]$  is set to  $-1$  when the two vertices are not adjacent.



**Fig. 1.** Example graph  $G$  and the corresponding hybrid representation.

Considering the 8-vertex graph  $G = (V, E)$ , the initial contents of  $AL$  and  $IM$  are shown in Figure 1. Note that  $AL$  is implemented using a two dimensional array for fast (direct) access via the indexing provided by  $IM$ . In general, we allocate enough memory to fit the neighbors of each vertex only. In addition to  $IM$  and  $AL$ , we introduce three linear arrays: the degree vector ( $DEG$ ), the vertex list ( $LIST$ ), and the vertex index list ( $IDXLIST$ ).

The degree vector holds the current neighborhood cardinality of each vertex,  $LIST$  contains the list of currently active (not deleted) vertices which we use instead of the degree vector for more efficient complete graph traversals, and

IDXLIST stores the index of each vertex in LIST. In other words,  $\text{LIST}[i]$  is the  $i^{\text{th}}$  vertex in the list of active vertices and  $\text{IDXLIST}[u]$  is the index of vertex  $u$  in LIST. All data structures except for the DEG vector are global and their memory is allocated at startup only. The DEG vector is local to every search-tree node (i.e., every recursive call, since a copy of the vector is passed as parameter).

In the next section, we show how the above structures are dynamically updated during a search algorithm, while performing various graph modification operations. We note that some operations, like edge contraction for example, require additional bookkeeping that we briefly describe later. However, most common operations can be performed using the five data structures described above, which when combined together form the (generic) hybrid graph representation.

## 4 Efficient Search Operations

The hybrid data structure was designed with an eye on efficient graph modification operations during backtracking. In addition, some basic operations like adjacency query and neighborhood traversal are also performed efficiently. For example, checking if two vertices  $u$  and  $v$  are adjacent takes  $\mathcal{O}(1)$  time: just check whether  $-1 < \text{IM}[u][v] < \text{DEG}[v]$ . Neighborhood search, on the other hand, runs in  $\mathcal{O}(d)$  time, where  $d$  is the current vertex degree.

Graph traversal is another frequent operation, often used to select a vertex or an edge having a certain property during branching algorithms. A complete graph traversal runs in  $\mathcal{O}(n_c)$  time as opposed to  $\mathcal{O}(n)$ , where  $n_c$  is the number of currently active vertices in the graph. This is possible because of our use of the LIST and IDXLIST vectors. In the rest of this section, we discuss how the hybrid method is used for graph modification operations.

### 4.1 Edge Deletion

The simplest and most frequent operation performed during the search is probably edge deletion. Maintaining the hybrid data structure in this case is straightforward. For deleting an edge  $(u, v)$ , the degrees of  $u$  and  $v$  are decremented by one and the adjacency lists of the two vertices are adjusted respectively by placing  $u$  at the last position of  $\text{AL}[v]$  and  $v$  at the last position of  $\text{AL}[u]$  (Figure 2). Each of these two operations consists of a single swap with the last element of the respective list, together with an adjustment of the positions in IM.

Going back to our illustrative graph  $G$ , after deleting edge  $(v_0, v_3)$ , the modified AL, IM, and DEG vectors will be as shown in Figure 3 (changes are shown in gray). No changes to LIST and IDXLIST are required for edge deletion since no vertices are removed from the graph.

Notice that this operation runs in  $\mathcal{O}(1)$  time as all the information required for switching positions in AL can be found in IM. Vertex  $v_3$  is no longer a neighbor of vertex  $v_0$  because  $\text{IM}[0][3] = 2$  which is not less than  $\text{DEG}[0] = 2$ .

---

```

1: procedure DELETEEDGE( $u, v$ )
2:    $i \leftarrow \text{IM}[v][u]$ ;
3:    $j \leftarrow \text{DEG}[u] - 1$ ;
4:    $x \leftarrow \text{AL}[u][j]$ ;
5:    $\text{AL}[u][i] \leftarrow x$ ;
6:    $\text{AL}[u][j] \leftarrow v$ ;
7:    $\text{IM}[x][u] \leftarrow i$ ;
8:    $\text{IM}[v][u] \leftarrow j$ ;
9:    $\text{DEG}[u] \leftarrow \text{DEG}[u] - 1$ ;
10:
11:   $i \leftarrow \text{IM}[u][v]$ ;
12:   $j \leftarrow \text{DEG}[v] - 1$ ;
13:   $x \leftarrow \text{AL}[v][j]$ ;
14:   $\text{AL}[v][i] \leftarrow x$ ;
15:   $\text{AL}[v][j] \leftarrow u$ ;
16:   $\text{IM}[x][v] \leftarrow i$ ;
17:   $\text{IM}[u][v] \leftarrow j$ ;
18:   $\text{DEG}[v] \leftarrow \text{DEG}[v] - 1$ ;
19: end procedure

```

---

**Fig. 2.** The DELETEEDGE procedure.

## 4.2 Undo Edge Deletion

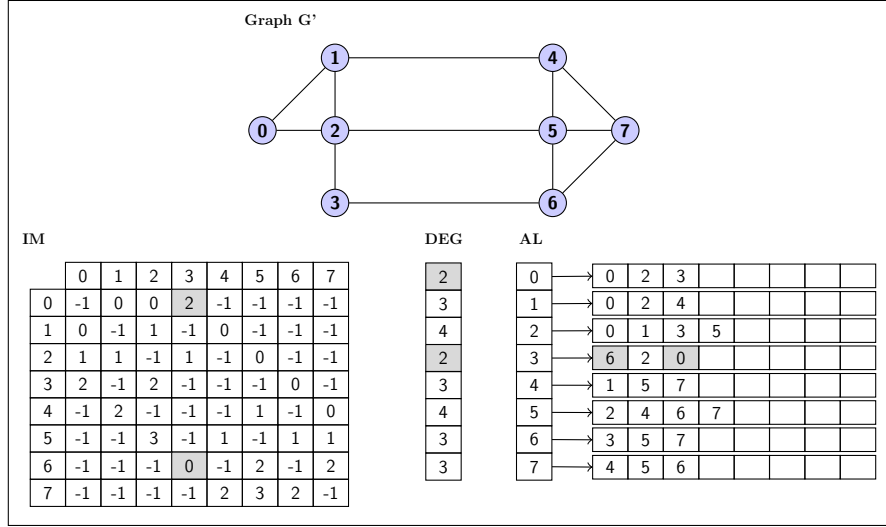
Now assume we want to undo the previous operation. This can be accomplished by simply setting  $\text{DEG}[0]$  and  $\text{DEG}[3]$  back to 3. There is no need to change  $\text{IM}$  and  $\text{AL}$  back to the previous state since no specific order is required for storing vertex neighbors. Moreover, since every search-tree node maintains its own copy of the  $\text{DEG}$  vector, there are actually no actions whatsoever that need to be taken for undoing edge deletion. Thus we have an *implicit-undo* for edge deletion operations. This implicit undo runs in  $\mathcal{O}(1)$  time.

## 4.3 Vertex Deletion

In a typical computation using adjacency lists, deleting a vertex  $v$  of current degree  $d$  requires traversing the list of neighbors of  $v$  and deleting  $v$  from the list of each of its neighbors. This operation runs in  $\mathcal{O}(\Delta(G)^2)$  time.

Using our hybrid computation, deleting a vertex  $v$  runs in  $\mathcal{O}(d)$  time where  $d$  is the (current) degree of  $v$ . This is simply performed by running the edge deletion operation for every active neighbor of  $v$  (Figure 4). In addition, we remove  $v$  from the list of active vertices by swapping it with the last active vertex in  $\text{LIST}$  and decrementing the number of active vertices by one. The  $\text{IDXLIST}$  plays the same role as  $\text{IM}$  for deleting a vertex from  $\text{LIST}$ .

To illustrate the purpose of the  $\text{LIST}$  vector, suppose we need to pick a vertex with a particular property, such as one of maximum degree (which is often needed for heuristic priorities).



**Fig. 3.** Hybrid representation after deleting edge  $(v_0, v_3)$ .

---

```

1: procedure DELETEVERTEX( $u, n_c$ )
2:    $d \leftarrow \text{DEG}[v]$ ;
3:    $last \leftarrow \text{LIST}[n_c - 1]$ ;
4:    $i \leftarrow \text{IDXLIST}[v]$ ;
5:    $\text{LIST}[i] \leftarrow last$ ;
6:    $\text{LIST}[n_c - 1] \leftarrow v$ ;
7:    $\text{IDXLIST}[last] \leftarrow i$ ;
8:    $\text{IDXLIST}[v] \leftarrow n_c - 1$ ;
9:   for  $j \leftarrow d - 1, 0$  do
10:     $u \leftarrow \text{AL}[v][i]$ ;
11:    DELETEEDGE( $u, v$ );
12:   end for
13: end procedure

```

---

**Fig. 4.** The DELETEVERTEX procedure.

Consider the two operations of copying the DEG vector and searching for the vertex of highest degree. Doing so would consume  $\Omega(n)$  time if the DEG vector is used alone. This is reduced to  $\mathcal{O}(n_c)$ , where  $n_c$  is the number of currently active vertices. To see why, note that iterating from  $i = 0$  to  $n_c$  only,  $\text{DEG}[\text{LIST}[i]]$  returns the degree of the vertex at position  $i$  in LIST. Knowing that a great majority of search tree nodes are near the bottom of a search tree, where the graph order is almost constant, this simple strategy alone makes a substantial difference.

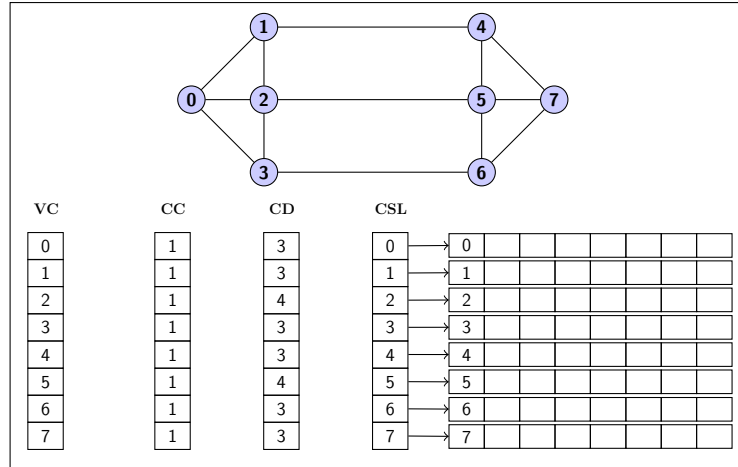
#### 4.4 Undo Vertex Deletion

As in the case of edge-deletion, the undo of vertex deletion is implicit (requires no action) when separate degree vectors are maintained at every search node. This is due to the fact that edge deletion can be considered as an independent operation. The only required operation is to increment the number of active vertices by one (if this value is stored globally) so that the LIST and IDXLIST vectors reflect the re-insertion of a vertex. Explicitly undoing a vertex deletion operation runs in  $\mathcal{O}(d)$  time where  $d$  is the degree of the deleted vertex (it mainly consists of incrementing the degree of each neighbor by one.)

#### 4.5 Edge Contraction

The next operation we consider is edge contraction. Contracting edge  $(u, v)$  replaces vertices  $u$  and  $v$  by a new vertex whose neighborhood is  $N(u) \cup N(v) \setminus \{u, v\}$ . To achieve our implicit-undo objective, we implement this operation using a coloring technique that requires additional bookkeeping. Simply, vertices with the same color are treated as one single vertex obtained by contracting edges between them. Initially, every vertex  $v_i$  is assigned color  $c_i$ , and every color class  $c_i$  has cardinality one and degree  $d(c_i) = d(v_i)$ . In addition to previously discussed data structures, we use the following (see Figure 5):

- VCOLOR vector: holds the current color of every vertex.
- COLOR\_CARD (CC) vector: holds the current cardinality of every color set.
- COLOR\_DEGREE (CD) vector: holds the current degree of every color set.
- COLOR\_SET\_LIST (CSL): holds the list of vertices belonging to every color set.

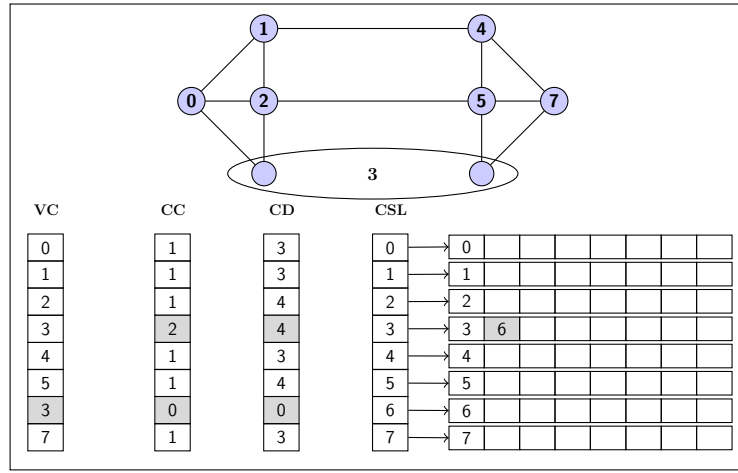


**Fig. 5.** Coloring data structures (previous data structures not shown but required).



Given that all operations now involve color sets, the LIST and IDXLIST do not hold vertex information anymore, but they maintain the list of active (not deleted) colors instead. If no edge contraction operations are performed, color sets would be identified by their corresponding vertices (or vice-versa).

When edge contraction is possible, the initial state of the graph  $G$  consists of all data structures previously discussed. AL, IM, CSL, LIST and IDXLIST would be globally stored (in RAM), while DEG, CD, CC and VCOLOR would be copied at every search-tree node. To contract an edge, say  $(v_3, v_6)$ , we actually assign both vertices the same color. Assuming we assign the two vertices color  $c_3$ , the required modifications are shown in Figure 6 (changes shown in gray).



**Fig. 6.** Coloring data structures after contracting edge  $(v_3, v_6)$  (modifications to previous data structures not shown here but required).

Since we are dealing with simple graphs, any edge between two vertices belonging to the same color set is deleted and no more than one edge is allowed between a color set and another. Clearly, such an operation can be implicitly taken back as well, but is also more time and space consuming than simple vertex deletion. This technique makes it possible to implement the vertex folding operation, introduced in [8], for the parameterized VC algorithm.

#### 4.6 The Vertex Cover Folding Operation

Let  $(G, k)$  be an instance of the VERTEX COVER problem and let  $u \in V(G)$  be a degree-two vertex with neighbors  $v$  and  $w$ . If  $v$  and  $w$  are adjacent, then there is a minimum vertex cover that contains  $v$  and  $w$  (and not  $u$ ). So it is safe to delete  $u$ , add  $v$  and  $w$  to the potential solution and decrement  $k$  by two. In the case where  $v$  and  $w$  are non-adjacent, an equivalent VERTEX COVER instance

is obtained by contracting edges  $uv$  and  $uw$  and decrementing  $k$  by one. This latter operation is known as degree-two vertex folding [8].

As we shall see, applying the coloring technique to implement vertex folding considerably improves the runtime on certain recalcitrant instances, but slows down the computation on graphs where folding rarely occurs. In such cases, the overhead of maintaining color-sets is a drawback. Note that folding alone made it possible to obtain a worst-case running time of  $\mathcal{O}^*(1.285^k)$  in [8]. Yet, our results show that excluding folding from the same algorithm is faster on a large number of instances, except for the case where the graph is regular (or nearly regular) where other reduction methods do not apply.

#### 4.7 Permanent Edge Addition

An edge addition performed at a search-tree node is considered permanent if no descendant search-tree node later modifies the operation (by edge or vertex deletion). In some problems, such as CLUSTER EDITING, searching for a solution involves edge deletions as well as permanent edge additions.

Using the classical adjacency list structure, it would be impossible to incorporate edge addition without extra bookkeeping to record changes to the AL. To incorporate the permanent edge addition operation to our hybrid data structure, we make the following modifications: (i) enough global memory is allocated to AL (i.e.  $n$ -by- $n$  matrix where  $n$  is the size of the input graph), and (ii) an additional degree vector (NDEG) is maintained at every search node.

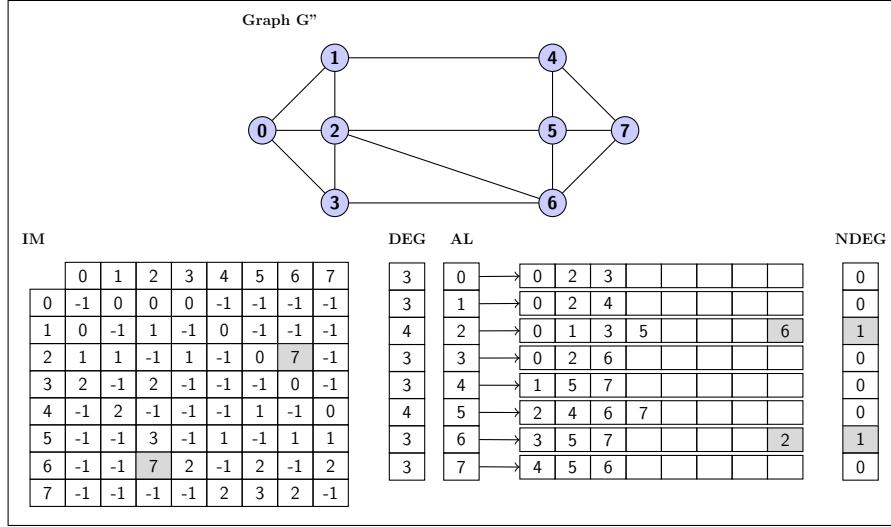
The additional memory requirement guarantees storage space and the NDEG vector serves as a second degree vector. Figure 7 illustrates the required changes to the extended hybrid data structure after adding edge  $(v_2, v_6)$  to our original graph  $G$ .

The adjacency query still takes  $\mathcal{O}(1)$  time but requires an additional condition:  $u$  and  $v$  are adjacent when  $-1 < \text{IM}[u][v] < \text{DEG}[v]$  or  $-1 < n - 1 - \text{IM}[u][v] < \text{NDEG}[v]$ . Neighborhood traversal becomes slightly more complicated but the time complexity remains unchanged. Undoing this operation only requires setting  $\text{NDEG}[2]$  and  $\text{NDEG}[6]$  back to 0 and subsequent addition of edges incident on  $v_2$  or  $v_6$  could overwrite older positions.

#### 4.8 Theoretical Runtime Analysis

Since we are dealing with exact algorithms for  $\mathcal{NP}$ -hard problems, our target is to achieve faster running times rather than accommodating large instances, which would not be possible to solve anyway. Yet, despite the apparent disadvantage of increasing the “global” space requirement, our algorithms do save the space needed for extra bookkeeping for explicit undo of graph modifications. In addition, we achieve constant-time implicit undo-operations which occur every time our recursive algorithms hit a backtracking state.

Table 1 summarizes the advantages of the hybrid graph representation over the adjacency list and adjacency matrix representations.



**Fig. 7.** Extended hybrid representation after adding edge  $(v_2, v_6)$ .

## 5 Experimental Results

Four different versions were implemented for the VERTEX COVER algorithm. AL\_VC\_OPT and HYBRID\_VC\_OPT are two generic search-tree optimization versions using the adjacency-list and hybrid graph representations respectively. In Table 2, the running times for both versions are reported for a number of DIMACS graphs.

HYBRID\_VC\_PARM is a parameterized hybrid version that does not take advantage of vertex folding, while HYBRID\_VCF\_PARM is a parameterized version implemented using the coloring technique described in the previous section for folding. In all the conducted experiments, the folding technique is at most two times slower than the simple generic branching algorithm. It gets faster as the difference between the highest and lowest vertex-degrees gets smaller. In particular, applying vertex folding via our coloring technique, is much faster on regular graphs. To illustrate, we report experiments on a well known 4-regular graph (the 120-Cell on 300 vertices), by varying the input parameter, and results are reported in Table 3.

As for the DOMINATING SET problem, AL\_DS\_OPT denotes the optimization version using the adjacency-lists representation and HYBRID\_DS\_OPT the optimization version using the hybrid graph representation. Running times on random graphs, with various densities, are given in Table 4.

To demonstrate the efficiency of the edge addition operation, we implemented two versions of the parameterized CLUSTER EDITING algorithm described in Section 2. AL\_CE\_PARM denotes the implementation using the adjacency-lists representation and HYBRID\_CE\_PARM refers to the version implemented with

**Table 1.** Runtime comparison for different search-operations using AL, AM, and the hybrid graph representation. We denote by  $n$  the number of vertices in the graph, by  $d(v)$  the degree of a vertex  $v$ , and  $d(c)$  denotes the degree of a color-set  $c$ .

Operation	Adjacency Matrix	Adjacency List	Hybrid
Adjacency Query	$\mathcal{O}(1)$	$\mathcal{O}(d(v))$	$\mathcal{O}(1)$
Neighborhood Traversal	$\mathcal{O}(n)$	$\mathcal{O}(d(v))$	$\mathcal{O}(d(v))$
Edge Deletion	$\mathcal{O}(1)$	$\mathcal{O}(d(v))$	$\mathcal{O}(1)$
Undo Edge Deletion	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Vertex Deletion	$\mathcal{O}(n)$	$\mathcal{O}(d(v)^2)$	$\mathcal{O}(d(v))$
Undo Vertex Deletion	$\mathcal{O}(d(v))$	$\mathcal{O}(d(v))$	$\mathcal{O}(1)$
Edge Contraction	$\mathcal{O}(n)$	$\mathcal{O}(d(v))$	$\mathcal{O}(d(c))$
Undo Edge Contraction	$\mathcal{O}(d(v))$	$\mathcal{O}(d(v))$	$\mathcal{O}(1)$
Edge Addition	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Undo Edge Addition	$\mathcal{O}(1)$	$\mathcal{O}(d(v))$	$\mathcal{O}(1)$

**Table 2.** AL\_VC\_OPT vs. HYBRID\_VC\_OPT (no folding).

Graph	$ V $	$ E $	$ C $	AL_VC_OPT	HYBRID_VC_OPT
p_hat500-1.clq	500	31569	450	3 hr 48 min	1 hr 23 min
p_hat700-1.clq	700	60999	635	> 1 week	93 hr 20 min
p_hat700-2.clq	700	121728	651	15 min 10 sec	3 min 44 sec
p_hat1000-2.clq	1000	244799	946	31 hr 26 min	5 hr 28 min
p_hat1500-3.clq	1500	847244	1488	20 min 57 sec	5 min 3 sec

the hybrid graph representation. The running times of the implementations on random graphs are shown in Table 5. The graphs were generated based on fixing the number of vertices, the number of clusters ( $C$  in the table below) and the parameter  $k$ . For each such triple, the vertices were distributed randomly over the  $C$  clusters, then intra-cluster and inter-cluster edges were deleted and added (respectively) randomly to obtain a CLUSTER EDITING instance.

All codes were implemented in standard C, and experiments were run on three types of conventional machines: Intel Core2 Duo 2.33 GHz, Intel Xeon Processor X5550 2.66 GHz Quad Core, and Intel Core i7-2600 3.40GHz Quad Core. However, the numbers reported in each row were obtained on the same architecture.

**Table 3.** HYBRID\_VC\_PARM vs. HYBRID\_VCF\_PARM (with folding) on a 4-regular graph having 300 vertices and 600 edges.

Vertex Cover Size ( $k$ )	Answer	No Folding	With Folding
190	yes	2 hr 14 min	6 min 27 sec
165	no	> 4 days	46 min 56 sec
160	no	38 hr 2 min	2 min 32 sec

**Table 4.** AL\_DS\_OPT vs. HYBRID\_DS\_OPT.

Graph	$ V $	$ E $	$ D $	AL_DS_OPT	HYBRID_DS_OPT
rgraph1	100	400	16	21 min 4 sec	4 min 11 sec
rgraph4	150	1200	14	16 hr 46 min	2 hr 27 min
rgraph5	150	1500	11	3 hr 31 min	28 min 20 sec
rgraph7	150	3000	7	27 min 16 sec	2 min 1 sec
rgraph8	200	4500	9	5 hr 44 min	30 min 8 sec
rgraph9	200	5000	8	1 hr 20 min	6 min 46 sec
rgraph10	200	6000	6	1 hr 36 min	7 min 13 sec
rgraph12	250	9000	8	14 hr 37 min	56 min 53 sec
rgraph13	250	10000	7	1 hr 30 min	5 min 34 sec
rgraph14	250	12000	5	4 hr 41 min	16 min 19 sec
rgraph17	300	22258	4	28 min 32 sec	1 min 10 sec
rgraph18	300	11063	8	133 hr 38 min	5 hr 54 min
rgraph19	300	11287	8	> 7 days	8 hr 14 min
rgraph21	1000	374552	3	28 min 37 sec	6 min 36 sec

**Table 5.** AL\_CE\_PARM vs. HYBRID\_CE\_PARM.

Graph	$ V $	$ E $	C	k	AL_CE_PARM	HYBRID_CE_PARM
rgraph1	125	1508	5	20	7 min 6 sec	2 min 12 sec
rgraph1	125	1508	5	25	4 hr 15 min	1 hr 12 min
rgraph1	125	1508	5	30	5 days 21 hr	1 day 2 hr
rgraph2	250	3010	10	20	8 min 54 sec	2 min 22 sec
rgraph2	250	3010	10	25	4 hr 46 min	1 hr 15 min
rgraph2	250	3010	10	30	6 day 9 hr	1 day 10 hr
rgraph3	500	6013	20	20	28 min 28 sec	8 min 46 sec
rgraph3	500	6013	20	25	15 hr 13 min	4 hr 25 min
rgraph3	500	6013	20	30	21 days 6 hr	4 days 2 hr
rgraph4	250	6139	5	20	28 min 31 sec	4 min 36 sec
rgraph4	250	6139	5	25	15 hr 13 min	2 hr 26 min
rgraph4	250	6139	5	30	20 days 14 hr	3 days 6 hr

## 6 Conclusion

We presented a hybrid graph representation that efficiently trades space for time and facilitates many common graph operations required during recursive backtracking. Experiments on VERTEX COVER, DOMINATING SET and CLUSTER EDITING showed the utility of using this dynamic data structure. The running times of the same algorithm were shown to be consistently reduced, sometimes from days to hours.

The hybrid method can also be applied to recursive enumeration problems that are based on search-tree algorithms, such as the well known Bron-Kerbosch algorithm for MAXIMAL CLIQUES ENUMERATION [6]. Some of the presented implementation techniques are used in our recent implementations of the algorithms described in [2].

The main focus in this paper was on graph modification operations that reduce the original graph size, such as vertex deletion and edge contraction. This can be applied for the implementation of branch-and-reduce algorithms. Operations that increase the size of a graph are harder to implement using the presented techniques. Vertex addition and non-permanent edge addition are notable examples that remain to be considered.

## References

1. F. N. Abu-Khzam. The multi-parameterized cluster editing problem. In P. Widmayer, Y. Xu, and B. Zhu, editors, *COCOA*, volume 8287 of *Lecture Notes in Computer Science*, pages 284–294. Springer, 2013.
2. F. N. Abu-khzam, N. E. Baldwin, M. A. Langston, and N. F. Samatova. On the relative efficiency of maximal clique enumeration algorithms, with application to high-throughput. In *Computational Biology, Proceedings, International Conference on Research Trends in Science and Technology*, 2005.
3. S. Böcker. A golden ratio parameterized algorithm for cluster editing. *J. Discrete Algorithms*, 16:79–89, 2012.
4. S. Böcker, S. Briesemeister, Q. B. A. Bui, , and A. Truss. Going weighted: Parameterized algorithms for cluster editing. *Theoretical Computer Science*, 410(52):5467–5480, 2009.
5. S. Böcker, S. Briesemeister, and G. W. Klau. Exact algorithms for cluster editing: Evaluation and experiments. *Algorithmica*, 60(2):316–334, 2011.
6. C. Bron and J. Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, Sept. 1973.
7. Y. Cao and J. Chen. Cluster editing: Kernelization based on edge cuts. *Algorithmica*, 64(1):152–169, 2012.
8. J. Chen, I. A. Kanj, and W. Jia. Vertex cover: Further observations and further improvements. *J. Algorithms*, 41(2):280–301, 2001.
9. J. Chen, I. A. Kanj, and G. Xia. Improved upper bounds for vertex cover. *Theor. Comput. Sci.*, 411(40-42):3736–3756, 2010.
10. J. Chen and J. Meng. A 2k kernel for the cluster editing problem. *J. Comput. Syst. Sci.*, 78(1):211–220, 2012.
11. F. V. Fomin, F. Grandoni, and D. Kratsch. Measure and conquer: Domination - a case study. In L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, editors, *ICALP*, volume 3580 of *Lecture Notes in Computer Science*, pages 191–203. Springer, 2005.
12. F. V. Fomin, F. Grandoni, and D. Kratsch. Measure and conquer: A simple  $o(2^{0.288n})$  independent set algorithm. In *IN PROC. 17TH SODA*, pages 18–25, 2006.
13. F. V. Fomin, D. Kratsch, and G. J. Woeginger. Exact (exponential) algorithms for the dominating set problem. In *In Proceedings of the Thirtieth International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 245–256, 2004.
14. J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Graph-modeled data clustering: Exact algorithms for clique generation. *Theory Comput. Syst.*, 38(4):373–392, 2005.
15. F. Grandoni. A note on the complexity of minimum dominating set. *Journal of Discrete Algorithms*, 4:209–214, 2006.
16. J. Guo. A more effective linear kernelization for cluster editing. *Theor. Comput. Sci.*, 410(8-10):718–726, 2009.

17. C. Komusiewicz and J. Uhlmann. Cluster editing with locally bounded modifications. *Discrete Applied Mathematics*, 160(15):2259–2270, 2012.
18. J. Rooij, J. Nederlof, and T. Dijk. Inclusion/exclusion meets measure and conquer. In A. Fiat and P. Sanders, editors, *Algorithms - ESA 2009*, volume 5757 of *Lecture Notes in Computer Science*, pages 554–565. Springer Berlin Heidelberg, 2009.
19. J. M. M. van Rooij and H. L. Bodlaender. Design by measure and conquer, a faster exact algorithm for dominating set. In *In Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 657–668, 2008.